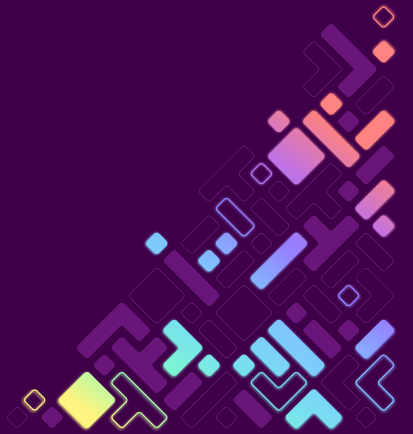
EMBEDDED LINUX CONFERENCE @ OPEN SOURCE SUMMIT EUROPE

THE LINUX FOUNDATION

Testing Your Yocto Project

From Ptest and Testimage to LAVA

Clara Kowalsky, Florian Bezdeka – Siemens AG



About us

Clara Kowalsky

<clara.kowalsky@siemens.com>

- Siemens Technology
- (In-house) Embedded Linux consultant & developer
- Contributor to major OSS projects for Siemens (Xenomai, isar, cip-core, OE-Core)

Florian Bezdeka

<florian.bezdeka@siemens.com>

- Siemens Technology
- (In-house) Embedded Linux consultant & developer
- Contributor to major OSS projects for Siemens (Xenomai, kernelci-core, isar, kas)

Test your project !

Image: Iconduck, Copilot

Agenda

testimage
Image tests

ptest
Package tests

1. Tutorial on setup and usage
2. Key learnings
3. Improvements brought upstream

LAVA
On-Device testing

1. Best practice sharing + Examples
2. Key learnings

Image: The Yocto Project, LAVA

Yocto testimage

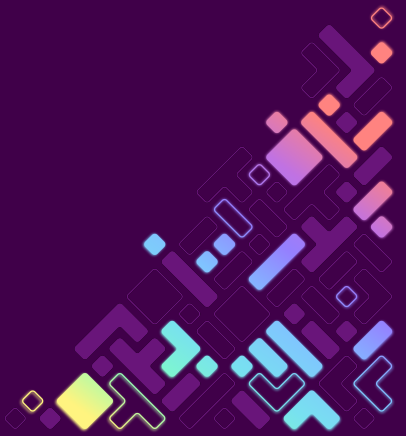


Image tests – testimage

- Project: The Yocto Project
- Doc: testimage wiki, testimage manual

- TL;DR: Run a series of automated tests inside your image
  - e.g., check network settings

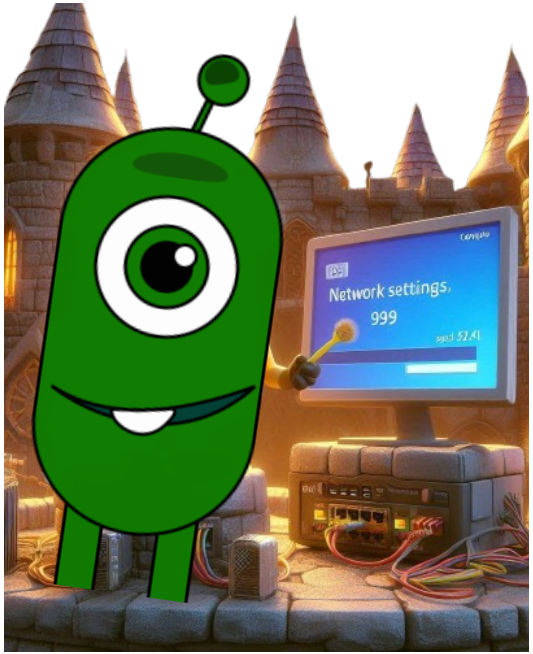


Image: Iconduck, Copilot

testimage – Key features

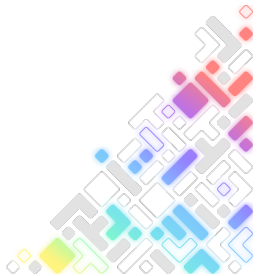
- Run image tests on QEMU or on the DUT (device under test)
- Consistent output:
  - `RESULTS - <testname>: <status> (<duration>)`
  - `PASSED` / `FAILED` / `SKIPPED`
- Results located in `build/tmp/log/oeqa`
  - `log.do_testimage`
  - `qemu_boot_log`
  - `testresults.json`

Image: Iconduck, Copilot

testimage – Available packages

- Yocto OpenEmbedded projects offer multiple testimage tests:
  - in `meta/lib/oeqa/runtime/cases`
  - e.g., OpenEmbedded-Core has 51 tests

- Customize testimage
  - In your project, add new tests to `<your_layer>/lib/oeqa/runtime/cases`



testimage – Example

From openembedded-core/meta/lib/oeqa/runtime/cases/python.py:

```
from oeqa.runtime.case import OERuntimeTestCase
from oeqa.core.decorator.depends import OETestDepends
from oeqa.runtime.decorator.package import OEHasPackage

class PythonTest(OERuntimeTestCase):
    @OETestDepends(['ssh.SSHTest.test_ssh'])
    @OEHasPackage(['python3-core'])
    def test_python3(self):
        cmd = "python3 -c \"...\""
        status, output = self.target.run(cmd)
        self.assertEqual(status, 0, msg='Exit status
   was not 0. Output: %s' % output)
```

Test is based on this class

Test depends on other tests

Runtime dependencies of the test

Test definition

testimage – Enable (1)

Example:

```
lib/oeqa/runtime/cases
├── fail.py
├── skip.py
└── success.py
```

- `IMAGE_CLASSES += "testimage"`
- Specify which tests to run:
  - `TEST_SUITES = "success skip fail"`
- For QEMU:
  - `TEST_TARGET = "qemu"`
  - Depending on architecture, adjust QEMU boot variables

Image: Iconduck, Copilot

testimage – Enable (2)

Example:

```
lib/oeqa/runtime/cases
├── fail.py
├── skip.py
└── success.py
```

- For hardware:
  - `TEST_TARGET = "simpleremote"`
  - `TEST_TARGET_IP = <IP_address_of_test_target>`
  - `TEST_SERVER_IP = <IP_address_of_test_server>`
- Build, install on target, connect via SSH and run

testimage – Run

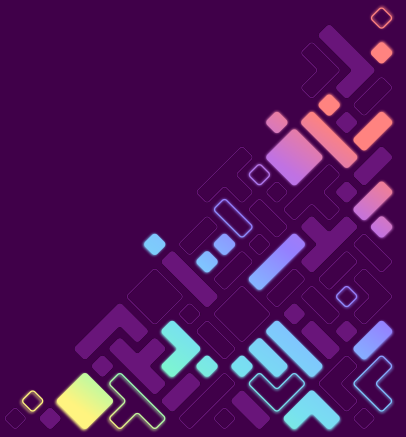
Example:

```
lib/oeqa/runtime/cases
├── fail.py
├── skip.py
└── success.py
```

- Run tests: **`bitbake -c testimage <target_image>`**

```
log.do_testimage:
...
RESULTS:
    RESULTS - success.SuccessTest.test_success: PASSED (0.65s)
    RESULTS - skip.SkipTest.test_skip: SKIPPED (0.00s)
    RESULTS - fail.FailTest.test_fail: FAILED (0.54s)
    SUMMARY:
    base-image () - Ran 3 tests in 1.186s
    base-image - FAIL - Required tests failed (successes=1,
    skipped=1, failures=1, errors=0)
```

Yocto ptest



Package tests – ptest

- Project: The Yocto Project
- Doc: ptest wiki, ptest manual

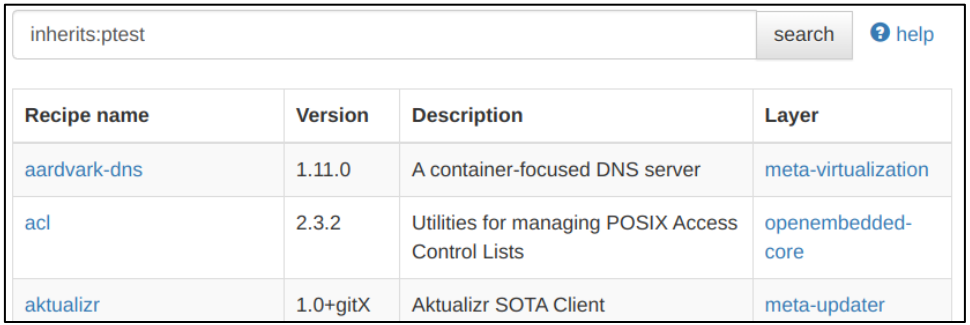
- TL;DR: Package and run the testsuites supplied in various upstream packages on the DUT

ptest – Key features

- Convention how things are packaged in Yocto (<pkg>-ptest)
- Consistent output:
  - `RESULTS - <testname>: <status> (<duration>)`
  - `PASSED / FAILED / SKIPPED`
- Automation:
  - Ptest-runner loops through ptest tests on the target
  - Script run-ptest starts the tests
- A ptest must contain: script run-ptest and the actual test

ptest – Available packages

- Yocto OpenEmbedded projects offer multiple packages with ptest
  - e.g., OpenEmbedded-Core has 114 ptests
  - Search for "inherit ptest" in OpenEmbedded



inherits:ptest search help

Recipe name	Version	Description	Layer
aardvark-dns	1.11.0	A container-focused DNS server	meta-virtualization
acl	2.3.2	Utilities for managing POSIX Access Control Lists	openembedded-core
aktualizr	1.0+gitX	Aktualizr SOTA Client	meta-updater

ptest – Example

From openembedded-core/meta/recipes-core/util-linux/util-linux_2.40.1.bb:

```
SRC_URI += "file://run-ptest"
inherit ptest
RDEPENDS:${PN}-ptest = "bash ..."
do_compile_ptest() {
    oe_runmake buildtest-TESTS
}
do_install_ptest () {
    mkdir -p ${D}${PTEST_PATH}/tests/ts
    ...
    cp -pR ${S}/tests/ts ${D}${PTEST_PATH}/tests/
}
```

Shell script that starts the test suite

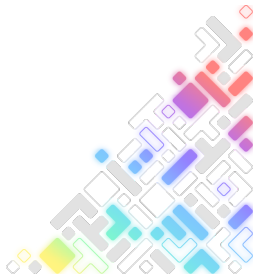
(Undetectable) runtime dependencies of the test

Compile the testsuite

Install test scripts

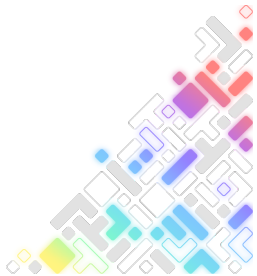
ptest – Enable (1)

- Build all -ptest packages:
  - `DISTRO_FEATURES:append = " ptest"`
- Install all -ptest packages in your image:
  - `EXTRA_IMAGE_FEATURES += "ptest-pkgs"`
- Install only specific -ptest packages:
  - `IMAGE_INSTALL += "util-linux-ptest <pkg-name>-ptest"`



ptest – Enable (2)

- Install ptest-runner:
  - `IMAGE_INSTALL += "ptest-runner"`
- All ptest files are installed in `/usr/lib/<pkg-name>/ptest`

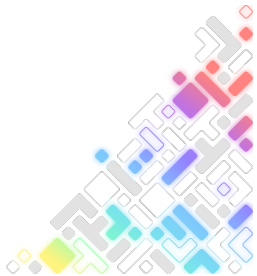


ptest – Run

- List available tests: `$ ptest-runner -l`
- Start all tests: `$ ptest-runner`
- Start specific test: `$ ptest-runner <pkg-name>`
- Run tests on the target: `$ ssh root@<target-IP> ptest-runner`

Run ptest in testimage:

- ptest depends on ping and ssh: `TEST_SUITES = "ping ssh ptest"`



ptest and testimage – Learnings

+ compile the testsuite ahead of time, no surprises at runtime about missing dependencies
+ well integrated into bitbake

- ~~large log output, can be difficult to interpret~~
- ~~result visualization needs postprocessing~~

Not anymore!

Improvement (1)

How to efficiently visualize the test results in the CI/CD pipeline?

- I wrote a script that generates a **unit test report in JUnit XML format**

  - Upstream in Yocto OpenEmbedded-Core project, commit 3f9be03
  - How to run the script:
    - `$ source oe-init-build-env`
    - `$ resulttool junit <testresults.json>`

Improvement (2)

How to efficiently visualize the test results in the CI/CD pipeline?

- I wrote a script that generates a **unit test report in JUnit XML format**

  - Tested with testimage and ptest results

```
build/tmp/log/oeqa
├── junit.xml
├── <target_image>
│   ├── log.do_testimage
│   └── qemu_boot_log
└── testresults.json
```

Run testimage in the GitLab CI/CD

- Target QEMU
- user-space networking (SLiRP)

Prework: enable SLiRP
Enable DHCP: `iface eth0 inet dhcp`

`<machine>.conf`: see

```
TEST_RUNQEMUPARAMS += "slirp"
QB_SLIRP_OPT = "-netdev user,id=net0,hostfwd=tcp:127.0.0.1:2222-:22,hostfwd=tcp:127.0.0.1:3000-:80"
```

`gitlab-ci.yml`: see Yocto doc runtime testing

```
test image:
  image: <kas:4.4>
  script:
    - kas build --target <target_image> base.yml
    - kas shell base.yml -c "bitbake -c testimage <target_image>"
```

Build your image: bitbake <target_image>

kas: Setup tool for bitbake-based projects

Run testimage

Run testimage in the GitLab CI/CD

- Target QEMU
- kernel-space networking (TAP)

Prework: enable TAP
Create tap: `ip tuntap add tap0 mode tap`
`<machine>.conf:`
`QB_TAP_OPT = "-netdev tap,id=net0,ifname=@TAP@,script=no,downscript=no"`

`gitlab-ci.yml`: see Yocto doc runtime testing

```
test image:
  image: <kas:4.4>
  script:
    - kas build --target <target_image> base.yml
    - kas shell base.yml -c "bitbake qemu-helper-native"
    - sudo ${CI_PROJECT_DIR}/poky/scripts/runqemu-gen-tapdevs 1000 1
    - kas shell base.yml -c "bitbake -c testimage <target_image>"
```

Required for setting up a tap interface

Create a tap interface

Run testimage in the GitLab CI/CD + Unit test report

`gitlab-ci.yml`: see Yocto doc runtime testing

```
...
  after_script:
    - kas shell base.yml -c
    "${CI_PROJECT_DIR}/poky/scripts/resulttool junit $
   {CI_PROJECT_DIR}/build/tmp/log/oeqa/testresults.json"
  artifacts:
    reports:
      junit:
        - ${CI_PROJECT_DIR}/build/tmp/log/oeqa/junit.xml
```

Call the script for unit test report generation

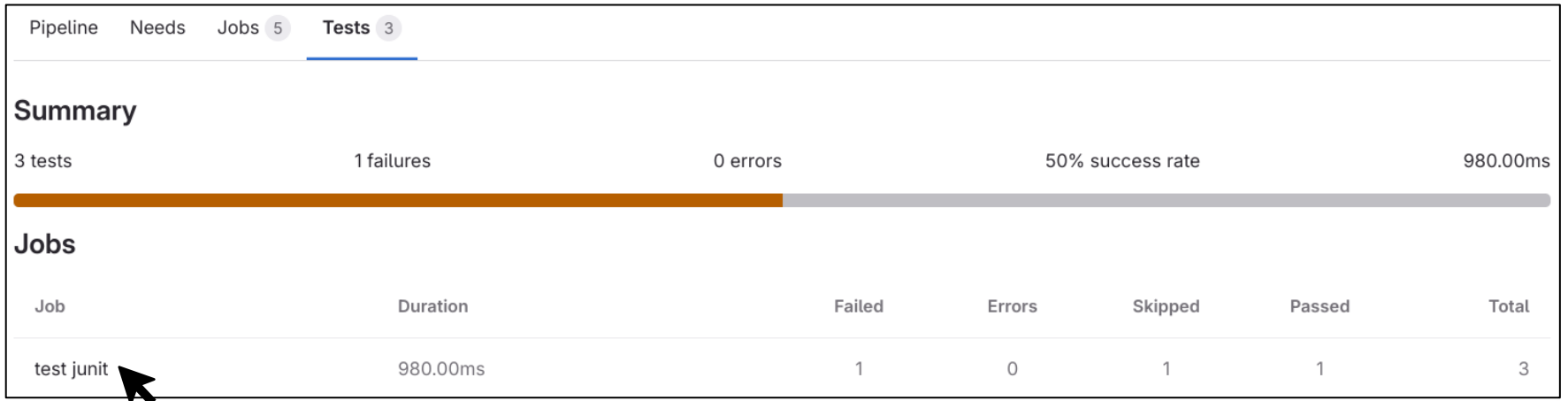
Visualize report in the CI/CD pipeline

Results in the GitLab CI/CD

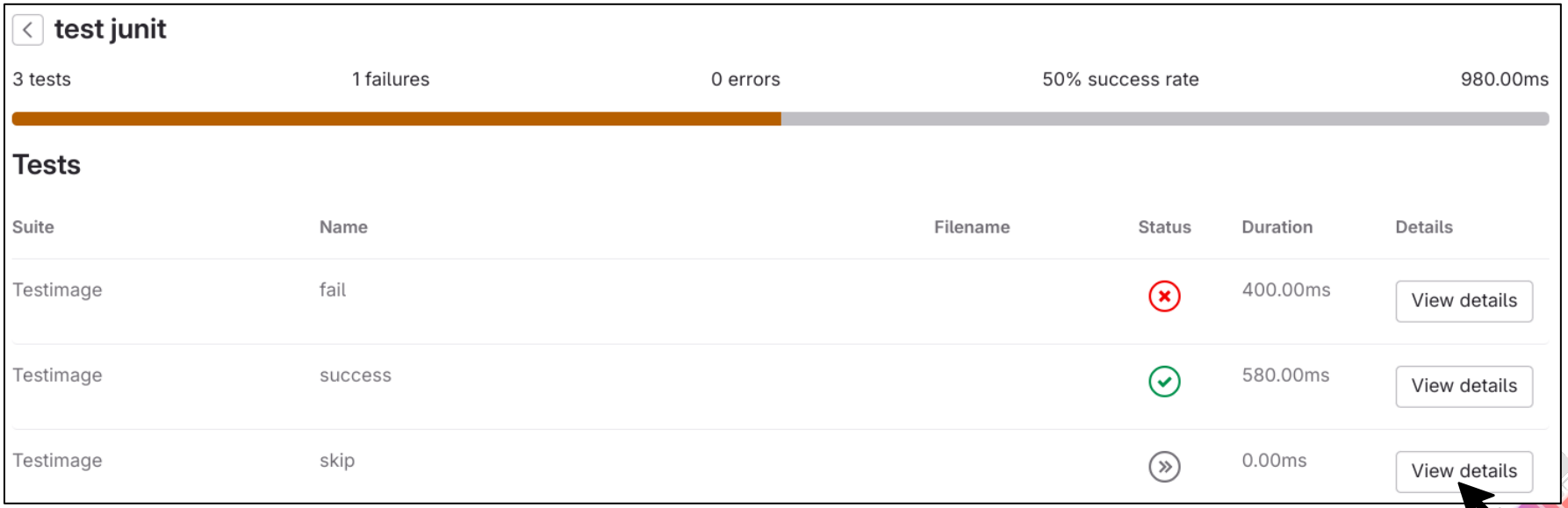
Example:

```
lib/oeqa/runtime/cases
├── fail.py
├── skip.py
└── success.py
```

- With: `TEST_SUITES = "success skip fail"`



Results in the GitLab CI/CD

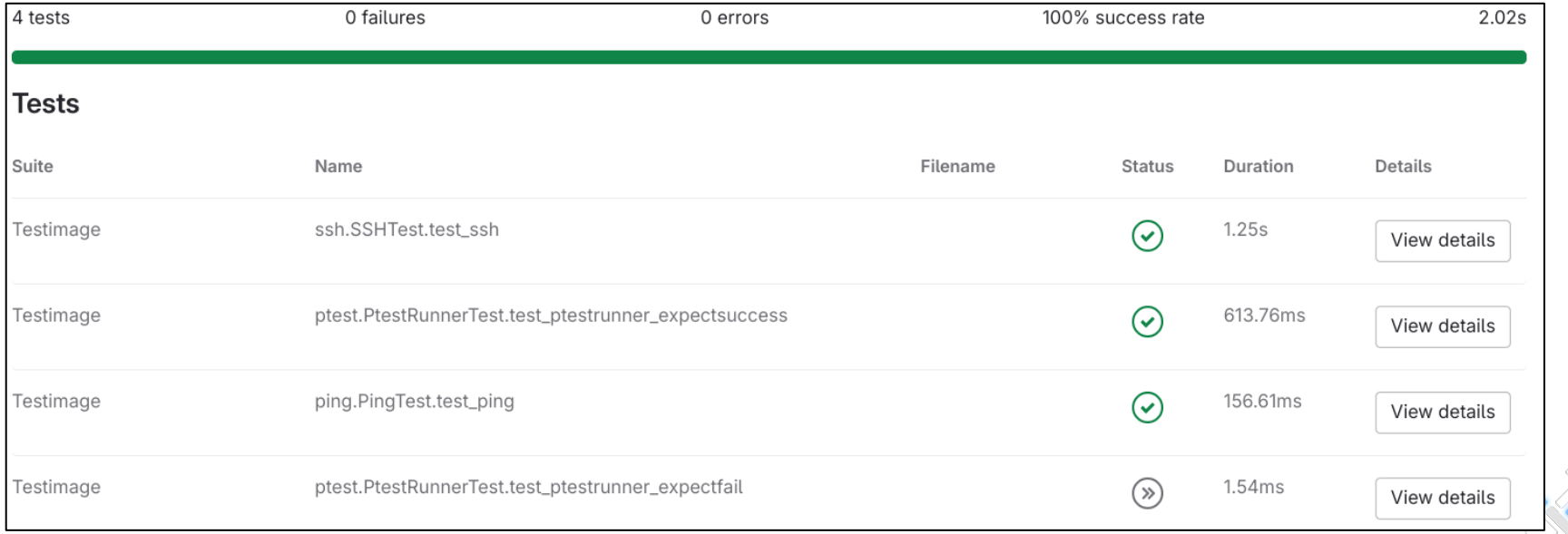


Results in the GitLab CI/CD

View details:

Results in the GitLab CI/CD

Also works with ptest: `TEST_SUITES = "ping ssh ptest"`



ptest and testimage – Learnings

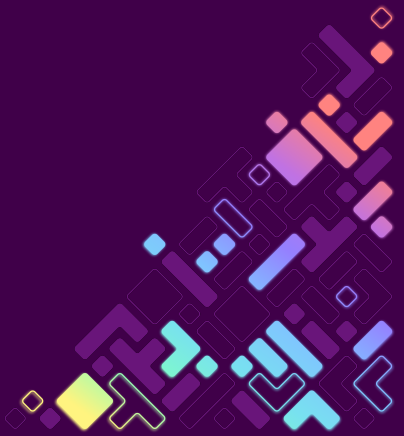
+ compile the testsuite ahead of time, no surprises at runtime about missing dependencies
+ well integrated into bitbake

- ~~large log output, can be difficult to interpret~~
- ~~result visualization needs postprocessing~~
- manual test execution necessary for DUT

Use LAVA instead?

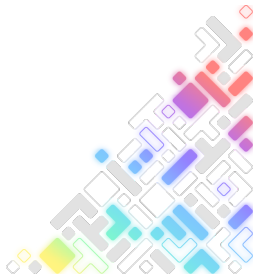
LAVA

(Linaro Automated Validation Architecture)



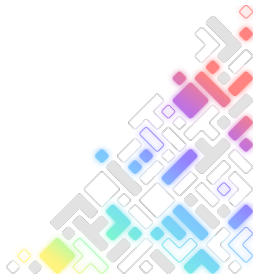
Why LAVA?

- With ptest and testimage we have some test infrastructure within our image
- How can we test that / run those tests on the target hardware?
- Maybe LAVA can help here...



Why LAVA fits for us

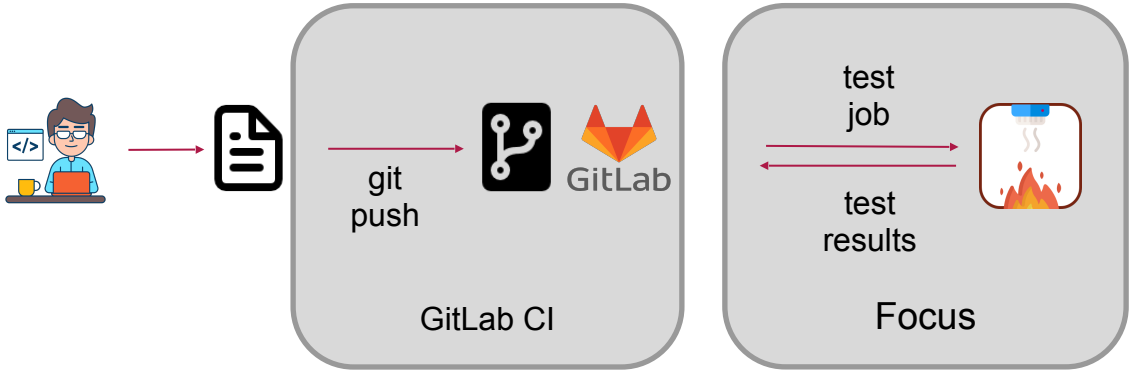
- On device testing required
  - Especially RT applications must be tested on the target platform
- Supported target hardware
  - Broad range of hardware (boards) supported / built in
  - Virtualized targets (qemu/kvm) are built in as well
- Infrastructure needs
  - Easy to maintain, Debian ships necessary packages
  - No special hardware requirements to the servers running LAVA
- Available as open source
- Easy integration into your development workflow



Development workflow

- Steps to get test results should be as simple as possible
  - Make necessary adjustments to bitbake/yocto recipes
  - Push changes into a development branch
  - Build and test are automatically scheduled
  - Notifications about the build/test results (via email)
  - Build and test results directly visible at the collaboration platform (GitLab)

Focus of today – LAVA, test job submission

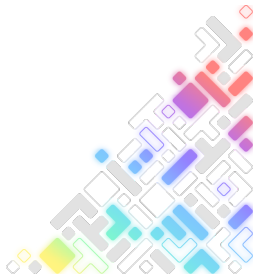


- Skipped
  - GitLab CI
    - Pipeline configuration
    - Necessary infrastructure
    - ...
- Focus
  - Introduction to LAVA
  - Job descriptions
  - Multinode device tests

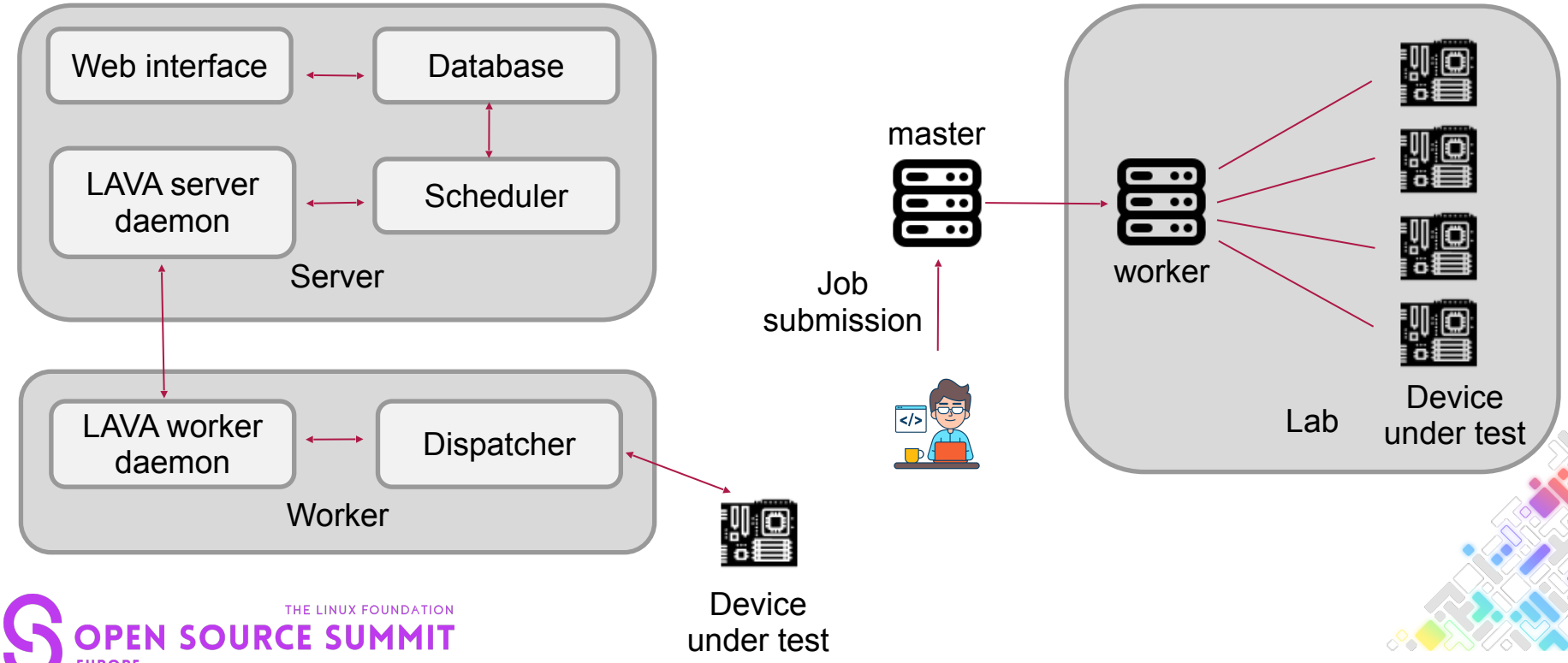
Image: Iconduck

LAVA – Linaro Automated Validation Architecture (1)

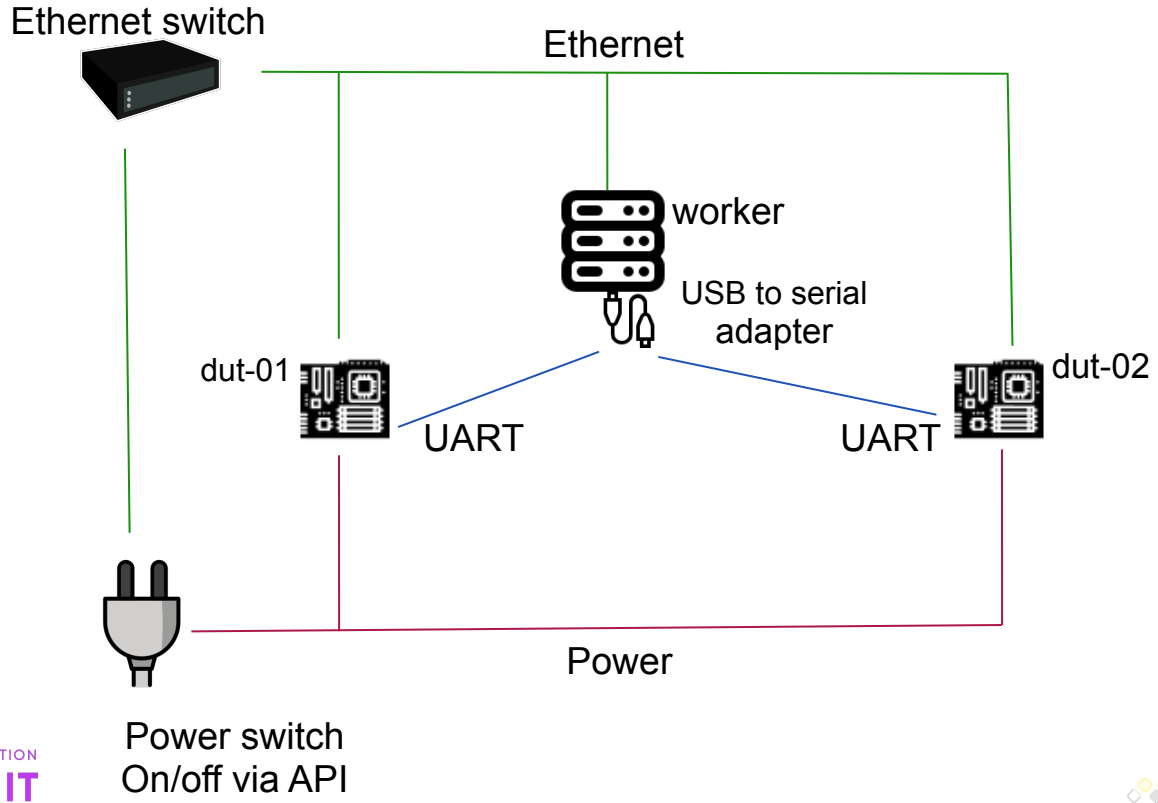
- You might know LAVA from
  - Linux Kernel Functional Testing (https://lkft.linaro.org/)
  - KernelCI (https://kernelci.org/)
- LAVA is a continuous integration system for deploying operating systems onto physical and virtual hardware for running tests
- LAVA is also used to manage and share boards among teams
- References
  - LAVA – Linaro Automated Validation Architecture (https://www.lavasoftware.org/)
  - LAVA development (https://gitlab.com/lava)



LAVA – Linaro Automated Validation Architecture (2)



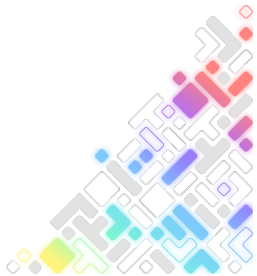
LAVA – Typical cabling in a lab



LAVA – A simple job description (1)

- Necessary information
  - Job name (job_name)
  - Device selection (device_type)
  - Three actions
    - Deploy action, where to fetch the kernel, ramdisk and rootfs from
    - Boot action, bootloader selection, how to boot up the device
    - Test action, definition of test commands and test suites

- In our example here
  - We download all artifacts (kernel image, initrd, rootfs) from our GitLab instance
  - Kernel and initrd are deployed to the tftp server running on the LAVA worker
  - Rootfs will be deployed to the NFS server on the LAVA worker



LAVA – A simple job description (2)

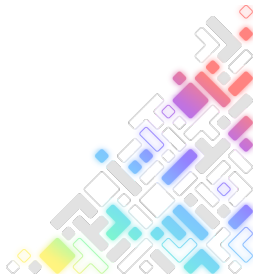
```
job_name: Your test job name / description
device_type: x86

actions:
  - deploy:
      to: tftp
      kernel:
        url: ${GITLAB_API}/artifacts/vmlinuz
       headers:
          PRIVATE-TOKEN: GITLAB_ARTIFACTS_AUTH_TOKEN
      ramdisk:
        url: ${GITLAB_API}/artifacts/initrd.img
      nfsrootfs:
        url: ${GITLAB_API}/artifacts/rootfs.tar.xz
        compression: xz
```

GITLAB_API="https://our.git.lab/api/v4/projects/${pid}/jobs/${jid}"

LAVA feature:
Remote artifacts auth token

LAVA – A simple job description (3)

```
- boot:
    method: ipxe
    commands: nfs
    prompts: ["root@mvs:"]
    auto_login:
      login_prompt: 'mvs login:'
      username: root
      password_prompt: 'Password:'
      password: root
```



LAVA – A simple job description (4)

```
- test:
      definitions:
        - name: your-testsuite
          from: inline
          path: inline/device.yml
          repository:
          metadata:
            format: Lava-Test Test Definition 1.0
            name: your-testsuite
          run:
            steps:
              - lava-test-case ptests --shell ptest-runner -t 600
              - lava-test-case pytests --shell pytest
```

Image: Iconduck

LAVA – Job submission

- Debian ships the lavacli package, a cmdline tool for job submission
- Typical steps
  - Configure (default) identitiy (authentication to lava instance)

```
lavacli identities add \
 --token ${your_token} \
 --uri ${lava_server_uri} \
 --username ${your_username} \
 default
```

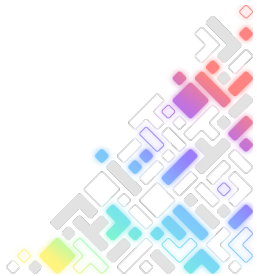
  - Submit job

```
lavacli jobs sumit ${path_to_job_description.yml}
```

  - Job ID will be printed to stdout

LAVA – Behind the scenes

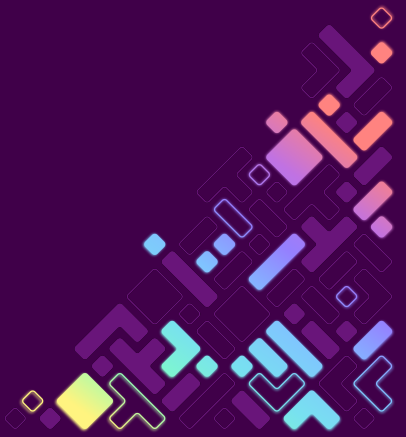
- Device Power on
- Boot phase
  - Interaction with the device / bootloader via UART
  - Deploy artifacts (kernel, initrd)
  - Interaction with the bootloader, might be in interactive mode
  - Login
- Test phase
  - Prepare execution of test suites
  - Run test suites
  - Collect test results (by reading stdout)
- Power off



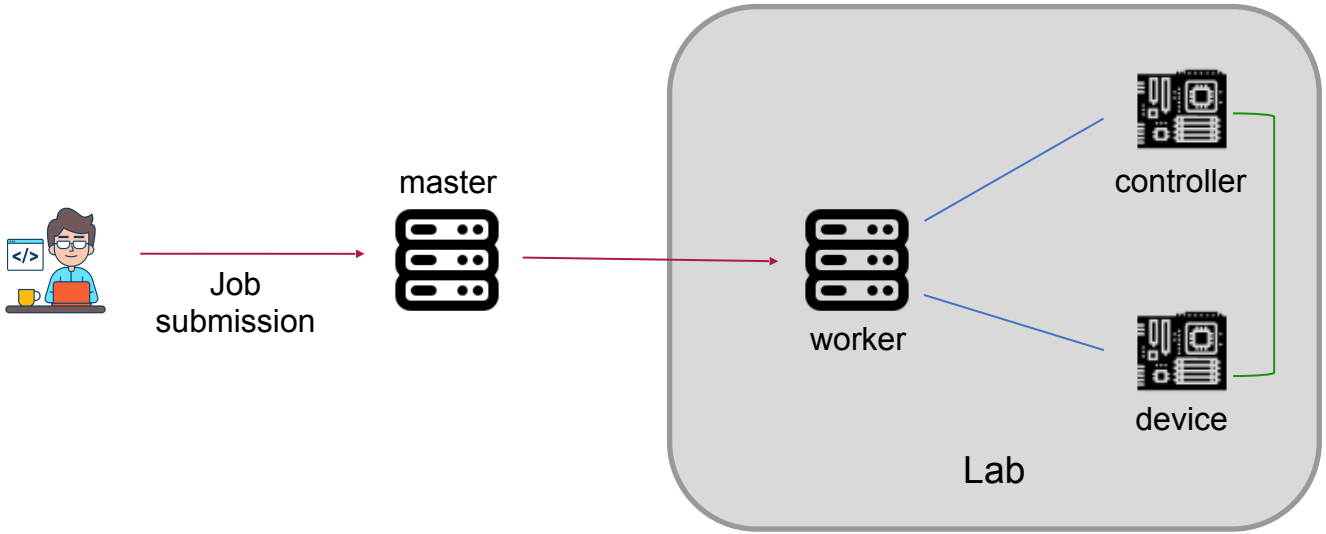
LAVA – GitLab integration of test results

- Fetch job logs via API
  `lavacli jobs logs ${job_id}`
- Fetch job results via API
  `lavacli results ${job_id}`
- Download JUnit test report (for public jobs without authentication)
  `curl https://${your_lava_master}/api/v0.2/jobs/${job_id}/junit`
- We integrated that into our GitLab CI, easily integrateable into other environments

LAVA – Multinode tests



LAVA – Multi node test (1)



LAVA – Multi node test (2)

```
protocols:
  lava-multinode:
    roles:
      controller:
        device_type: x86
        count: 1
      device:
        device_type: x86
        count: 1
```

LAVA – Multi node test (3)

```
actions:
...
 - test:
    role:
      - controller
    definitions:
      - name: controller-specific-testsuite
        from: inline
        path: inline/controller.yml
        repository:
          metadata:
            format: Lava-Test Test Definition 1.0
            name: controller-testsuite
          run:
            steps:
              - lava-send controller_ready
```

cross-device synchronization

LAVA – Multi node test (4)

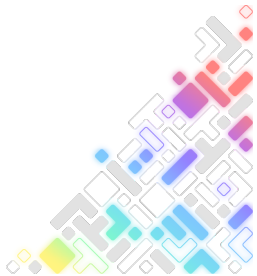
```
actions:
...
 - test:
    role:
      - device
    definitions:
      - name: device-specific-testsuite
        from: inline
        path: inline/device.yml
        repository:
          metadata:
            format: Lava-Test Test Definition 1.0
            name: device-testsuite
          run:
            steps:
              - lava-wait controller_ready
```

cross-device synchronization

LAVA – Learnings

- It takes some time to set it up
- A couple of infrastructure components must work closely together
- Configuration efforts

+ It scales!
+ Great way of sharing boards / hardware within one organization
+ Independent from the build chain (Yocto, …) of your project
+ Quite easy to integrate into best practice workflows
+ Quite easy to maintain (Debian ships all necessary packages)



Summary – Thank you. Questions?